

## SANDIA REPORT

SAND2017-5220  
Unlimited Release  
Printed May 2017

# Scalability of Several Asynchronous Many-Task Models for *In Situ* Statistical Analysis

Philippe P. Pébaÿ, Janine C. Bennett, Hemanth Kolla, Giulio Borghesi

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multitechnology laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-Mail: [reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)  
Online ordering: <http://www.osti.gov/bridge>

Available to the public from  
U.S. Department of Commerce  
National Technical Information Service  
5285 Port Royal Rd  
Springfield, VA 22161

Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-Mail: [orders@ntis.fedworld.gov](mailto:orders@ntis.fedworld.gov)  
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



# Scalability of Several Asynchronous Many-Task Models for *In Situ* Statistical Analysis

Philippe P. Pébay, Janine C. Bennett, Hemanth Kolla, Giulio Borghesi  
Sandia National Laboratories  
P.O. Box 969  
Livermore, CA 94551, U.S.A.  
{pppebay, gborghe, hnkolla, jcbenne}@sandia.gov

## Abstract

This report is a sequel to [PB16], in which we provided a first progress report on research and development towards a scalable, asynchronous many-task, *in situ* statistical analysis engine using the Legion runtime system. This earlier work included a prototype implementation of a proposed solution, using a proxy mini-application as a surrogate for a full-scale scientific simulation code. The first scalability studies were conducted with the above on modestly-sized experimental clusters.

In contrast, in the current work we have integrated our *in situ* analysis engines with a full-size scientific application (S3D, using the Legion-SPMD model), and have conducted numerical tests on the largest computational platform currently available for DOE science applications. We also provide details regarding the design and development of a light-weight asynchronous collectives library. We describe how this library is utilized within our SPMD-Legion S3D workflow, and compare the data *aggregation* technique deployed herein to the approach taken within our previous work.

## Acknowledgments

The authors would like to thank Sean TREICHLER at NVIDIA for his invaluable help in the context of this work, and Elliott SLAUGHTER at Stanford for having provided us with the most recent information regarding the SMPD-style of Legion programming.

# Contents

1	Introduction .....	7
1.1	About this Work .....	7
1.2	Outline .....	8
2	Background .....	10
2.1	Data-centric AMT Approach Benefits .....	10
2.2	The Legion Data-centric AMT Model and Run-time .....	10
2.3	SPMD Parallel Statistics Toolkit .....	11
2.4	Parallel Update Formulas .....	14
2.5	Evaluation Criteria .....	16
3	Parallel Statistics in Legion: Aggregation Regions Approach .....	18
3.1	Overview of the Aggregation Regions Approach .....	18
3.2	<i>In Situ</i> Setting with MiniAero .....	21
3.3	Aggregation Regions Results .....	24
4	Parallel Statistics in Legion: the Legion-SPMD Approach .....	26
4.1	Overview of the Legion-SPMD Approach .....	26
4.2	<i>In Situ</i> Setting with S3D .....	29
4.3	Test Platform .....	30
4.4	Results .....	30
5	Conclusion .....	34
	References .....	37

This page intentionally left blank

# 1 Introduction

As we look ahead to next generation high performance computing platforms, the placement and movement of data is becoming the key-limiting factor on both performance and energy efficiency. Furthermore, the increased quantities of data the systems are capable of generating, in conjunction with the insufficient rate of improvements in the supporting input/output (I/O) infrastructure, is forcing applications away from the off-line post-processing of data towards techniques based on *in situ* analysis and visualization. Together, these challenges are shaping how we will both design and develop effective, performant and energy-efficient software. In particular, the challenges highlight the need for data and data-centric operations to be fundamental in the reasoning about, and optimization of, scientific workflows on extreme-scale architectures.

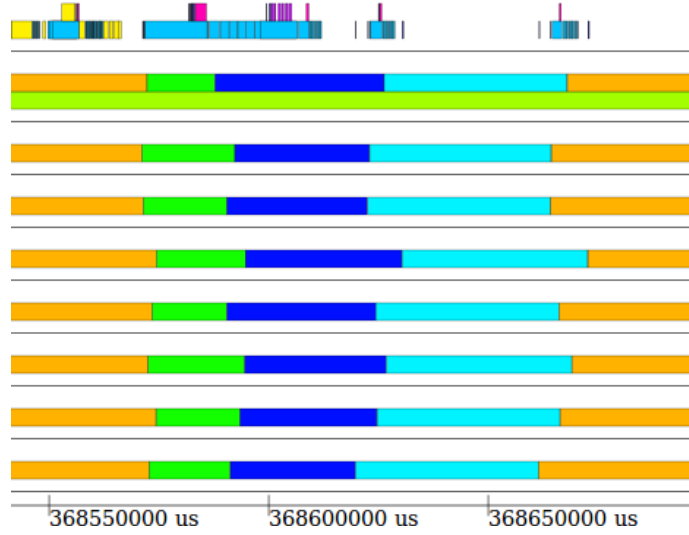
The DOE ASCR-funded project titled, “A Unified Data-Driven Approach for Programming In Situ Analysis and Visualization” (UDDAP), seeks to understand the interplay between data-centric programming model requirements at extreme-scale and the overall impact of those requirements on the design, capabilities, flexibility, and implementation details for both applications and supporting *in situ* infrastructure. The project leverages the Legion programming model and run-time system [BTSA12], that was developed as part of the ExaCT co-design center. In this report, we summarize recent research contributions from the Sandia sub-team of UDDAP project.

## 1.1 About this Work

This report extends the work of [PB16], in which we summarize research and development of statistical analysis kernels for *in situ* deployment within the Legion run-time. We note that porting the statistical analysis code from an MPI implementation into a Legion one was straightforward, thanks in part to the initial design of our parallel statistics packages that separated computation from communication, cf. [PTBM11], a design pattern which we discuss in detail below in §3.1.

The overall performance of the *in situ* system was evaluated on a number of small-size test beds using a computational fluid dynamics (CFD) mini-application called MiniAero, cf. [FFL15]. This implementation corresponded to a model first described in in [PB15], where we remarked that our approach, based on *aggregation regions* as surrogates for bulk-synchronous collective operations, was exhibiting optimal on-node parallel scaling, thereby taking advantage of the multiplicity of cores on each node.

We extended these early results in a more systematic study [PBH<sup>+</sup>16], which both confirmed our earlier encouraging results and also evinced that the additional overhead from the inclusion of the statistics computations *in situ* would not become a bottleneck. This additional finding was made possible by the instrumenting of the code using the Legion profiling tools, as illustrated in Figure 1. One of the primary benefits of the approach presented was that only a small set of well contained code was required to connect the analysis to the main simulation application. However, given that the findings in these earlier reports did not stem from a full-scale scientific code, it remained to be proven whether the nice portability features of our approach would extend to this



**Figure 1.** Task scheduling timeline of *in situ* analysis in MiniAero-Legion: Each row shows color-coded tasks running on a different CPU core. Learn sub-tasks are displayed in light blue color; other colors correspond to simulation tasks.

more complex setting.

## 1.2 Outline

This report briefly summarizes prior work, and presents an entirely new implementation of the parallel descriptive analysis engine that:

1. uses a full-scale simulation code as the main application driver, and
2. employs a variant of the Legion programming model that scales to full-scale simulation runs.

Namely, we use S3D, a massively parallel DNS solver developed at Sandia National Laboratories, cf.[HSSC05], for which a Legion implementation already exists and is routinely run on the largest production platforms available at DOE. Notably, the implementation of S3D relies on the SPMD (Single Program, Multiple Data) style of Legion, which is currently seen as a practical way to achieve scalability, in particular for applications based on main tasks that exist throughout all or most of the run. This style of Legion is based on *dynamic collectives*, i.e., non-blocking barriers advanced on a specified number of *arrivals*. These dynamic collectives allow the developer to define reduction objects<sup>1</sup> when programming in the SMPD style of Legion, provided the desired reduction operations be associative and commutative. We deploy our statistical analysis engine

<sup>1</sup>specifically, implemented as *fold* method of a dynamic collective.

within an SPMD-Legion S3D workflow and compare this implementation with that of aggregations regions, which is better suited to the fully-hierarchical variant of the Legion run-time.

We begin by providing brief overviews of the Legion asynchronous many-task (AMT) programming system and our parallel statistical analysis system design. We then summarize the *aggregation region*-based statistical analysis engines, providing a brief summary of scalability results up to 256 CPUs distributed across as many nodes. We follow this with a complete description of the statistics engine written using the SPMD-Legion model along with a scalability study illustrating the efficacy of the approach. A key component of the SPMD-Legion *in situ* analytics work has been the development of a light-weight asynchronous collectives library. We describe the library herein, highlighting the potential for its impact across a wide variety of application use cases.

## 2 Background

### 2.1 Data-centric AMT Approach Benefits

In a conventional *in situ* framework, computer simulation programs and analysis code bases must be explicitly connected, requiring manual data management and communication. Subsequent changes in how analysis is done thus requires rewriting parts of the simulation, and vice-versa. One of the key advantages of a data-centric AMT model is that it naturally supports composability of simulation and analysis code bases, thereby reducing the entanglement of the application and analysis codes to simply *what* data is being shared, and not *when*, *where*, or *how* this sharing is to occur. Analysis tasks should therefore be much easier to incorporate into simulation code, and re-usability by other codes should also be vastly improved.

In addition, the AMT approach provides performance portability in the face of increasing I/O cost and variability. Different pieces of code will likely have different spatio-temporal characteristics in terms of compute intensity, degree of parallelism, data access patterns, and task and data inter-dependencies. The decoupling of the functional code from computation and data placement allows an analysis code to easily be tuned for different machines or to be easily implemented within either *in situ* or *in transit* frameworks.

Finally, a data-centric AMT approach provides an opportunity for the run-time to efficiently co-schedule simulation and analysis tasks. It may be possible to incorporate the analysis workload into available gaps in the execution of the simulation, whereas this scheduling requires significant programmer effort (and is generally not performance-portable) in the MPI+X models, cf. [Gro09]. Dynamic load balancing provided by AMT models has the potential to allow for more graceful handling of dynamic variability in analysis tasks and simulation.

### 2.2 The Legion Data-centric AMT Model and Run-time

Legion (cf. [BTSA12]) is an AMT model that makes data and data-centric operations first-class programming constructs. A Legion application is decomposed into a task hierarchy, where tasks declare which parts of the application data they will access or update. The Legion model separates the functional description of the code (i.e., tasks and the data upon which they operate) from the way in which it is mapped to a given machine (i.e., tasks and data placement). Application data is contained in *logical regions*, which have neither an implied location within the memory hierarchy of the machine nor a fixed physical layout.

The Legion run-time leverages these data properties to issue data movement operations as needed, removing this burden from the developer. This run-time system detects pairs of tasks that have a data dependence (i.e., they may access the same data and at least one is making non-commutative modifications to it) and guarantees that the second task in the pair does not execute until it is safe to do so. This technique extracts (dynamic) task parallelism from the application, while preserving programmer-friendly “apparently-sequential” execution semantics. Legion

run-time calls are thus deferred as needed, allowing the application code to issue tasks with dependencies immediately.

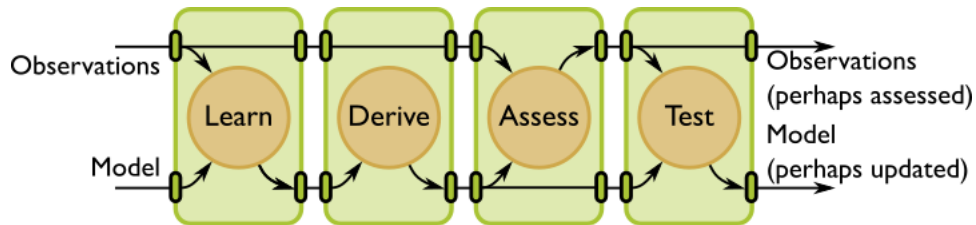
In addition, this approach also allows for the dynamic execution of performance-related transformations (e.g., the replication of read-only data to increase parallelism), perhaps differently on different machines, without modifying the “machine-agnostic” functional description. Such run-time transformations also have the potential to alleviate the risks of early optimization as well as the many burdens of late-stage performance analysis that are commonplace in the bulk-synchronous SMPD context.

Legion is designed for two classes of users: advanced application developers and domain-specific (DSL) and library authors. It is therefore a natural candidate as an AMT run-time system to support the requirements outlined in §1.

## 2.3 SPMD Parallel Statistics Toolkit

The prevalence of large, distributed data sets has lead to the development of statistical packages to analyze them in parallel, cf. [WBS08, WTP<sup>+</sup>08, SME<sup>+</sup>09, PTBM11, Edd10, Sta10]. In [PTBM11] we have presented a comprehensive view of the scalable, parallel statistical analysis library which we designed and implemented and released as part of VTK. Without providing a comprehensive view of this library, which would be beyond the scope of this report, we hereafter justify the choice it as a first candidate for our research on AMT in-situ analysis using Legion.

The parallel statistics *engines* of VTK are a set of C++ classes, developed at Sandia National Laboratories between 2008 and 2013, in particular by the authors of this report, based on SPMD data parallelism using MPI. We therefore knew this tool kit very well before starting this study; we knew in particular the limitations of the bulk-synchronicity when attempting to achieve extreme-scale scalability. There was therefore a strong motivation to research other approaches, besides SPMD, which would allow for data analysis at scales orders of magnitude beyond what we were able to achieve with MPI (optimal scalability with up to  $O(10^5)$  cores).



**Figure 2.** The 4 operations of statistical analysis and their interactions with input observations and models. When an operation is not requested, it is eliminated by connecting input to output ports.

The design decisions made during development were motivated by two primary factors:

- (i) We wanted to mimic the predominant types of data analysis work flows, so that a data analyst using our framework would find it natural and intuitive to use.
- (ii) The design had to be conducive to embarrassingly parallel implementations when possible.

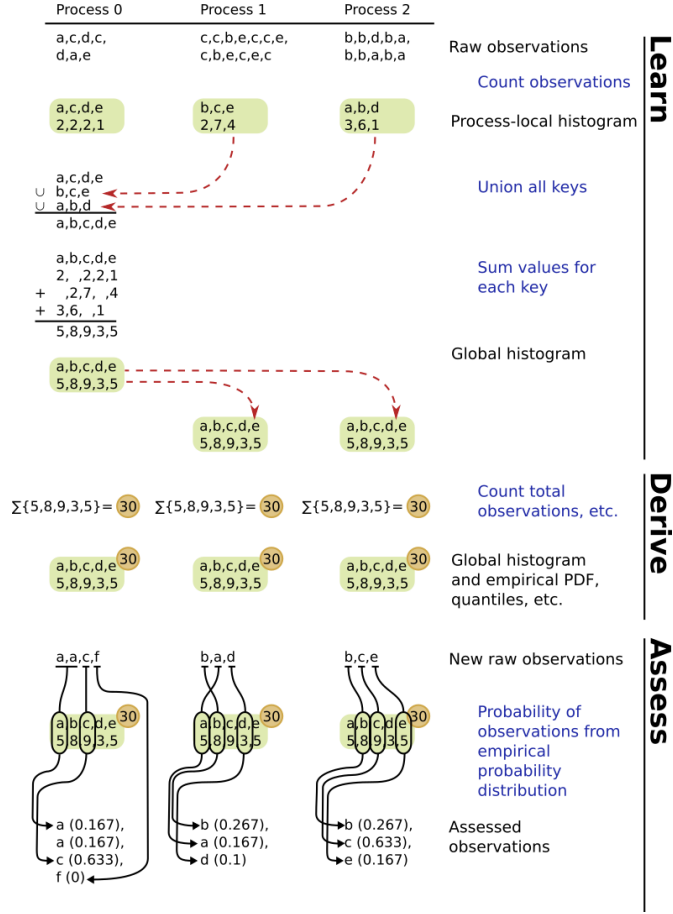
In order to meet these two overlapping but not exactly congruent design requirements, we isolated those parts of the analysis which by construction are not embarrassingly parallel (due to the mathematics of the statistical analysis itself, not due to our design) so that parallel design trade-offs be limited to those components where embarrassingly parallel implementations are not viable. Specifically, the statistical analysis work flow is split into 4 disjoint operations:

- Learn a model from observations,
- Derive statistics from a model,
- Assess observations with a model, and
- Test a hypothesis.

These operations, when all are executed, occur in order as shown in Figure 2. However, it is also possible to execute only a subset of these, for example when it is desired that previously computed models, or models constructed with expert knowledge, be used in conjunction with existing data. Note that in earlier publications, only the first 3 operations were mentioned; the Test operation, which we initially saw as a part of Derive, was separated out afterwards for reasons both theoretical (statistical hypothesis testing relies on assumptions not accepted by all statisticians) and practical (second pass through the data might be necessary, as well as calls to outside libraries). We therefore always considered the Test operation as an experimental feature, not implemented for all statistics engines, and therefore we leave it aside for the current study.

From the parallelism standpoint, this subdivision of the work flow reduces the Learn operation to a special case of the map-reduce pattern [DG04], while the remaining two are embarrassingly parallel. Specifically, the Learn operation belongs to the map-reduce pattern in that the parallel algorithm computes (maps) a set of distributed (key,value) pairs. All local values associated with the same key are then merged by the reduce function to compute the global *primary model*. In some of our statistical algorithms, namely *moment-based*, it is not necessary to communicate the keys for there is a fixed number of them, identical across all processes, and these keys may be ordered uniquely, so sending values alone is unambiguous.

In addition, the number of such keys is typically very small (less than 10), which allowed us to implement the reduce function as an AllGather MPI collective, allowing each core to perform locally all subsequent operations with no further communication. In contrast, for *quanta-based* algorithms, tables with an arbitrary number of key-value pairs must be communicated and different keys may be present on each process. We therefore implemented the reduce function using a Gather-Broadcast two-step operation, involving a small number of reduction nodes on which this procedure is performed, as illustrated in Figure 3 with a univariate analysis where the Learn operation builds a global histogram, along with derived statistics such as empirical PDF and quantiles.



**Figure 3.** A simplified example illustrating the operations of the parallel order statistics; dashed red arrows indicate inter-process communication. In terms of the map-reduce pattern, keys are the raw observations (represented by letters a, b, c, d, e) and values are the number of observations.

This can potentially cause problems as the size of the network increases to peta-scale, thereby justifying the need to investigate asynchronous reduction strategies.

We conducted several systematic, multiple-parameter studies of the scalability properties of our SPMD/MPI implementation with up to  $10^5$  cores on a tera-scale system, studying both weak and strong scaling properties. Those tests demonstrated, in particular, that our design allowed for optimal scalability of all moment-based engines, provided the input data sets were evenly distributed across the system. Note that these studies purposefully ignored I/O and data movement issues, using instead synthetic test data sets were generated on-the-fly by each process participating in the experiment in order to isolate the analysis per se from other aspects that mostly depend on the target platform. On the other hand, we established that for quanta-based statistics, our design trade-offs (between efficiency and robustness) allows for optimal strong and weak scaling when those statistical techniques are used in their appropriate context, namely, when the input data is not

quasi-diffuse, but honestly represents discrete measurements. Furthermore, those parallel engines have been successfully applied to the analysis of large-scale turbulent, reacting flow simulation data [BGP<sup>+</sup>09].

## 2.4 Parallel Update Formulas

Central moments, including the variance, and derived quantities like skewness and kurtosis, are some of the most widely used tools in descriptive statistics. However, standard approaches for computing them either require two passes over the data, or are grossly inaccurate for data that is not contained within a very limited range. This poses a problem in streaming settings where incremental results are needed after each new value is observed, and for very large datasets, which may not fit in available memory, and increasingly are distributed over a number of hosts.

In this setting the cost of distributed memory access is so large that two-pass algorithms become entirely impractical. Even a single machine increasingly performs large parallel computations on a Graphics Processing Unit (GPU), where memory bandwidth is a significant bottleneck. Using two passes doubles the execution time, and using double precision arithmetic doubles it again, almost irrespective of the number of arithmetic operations performed in each pass.

As the order of the moment increases, even the venerable two-pass algorithm may be inaccurate, as the numerical error for evaluating polynomials around the mean grows exponentially with the degree. When communication costs are the bottleneck, doubling the working precision doubles the computation time. Alternatives, such as compensation algorithms for summation (cf.[ORO05]) and polynomial evaluation (cf.[LL07]), require twice as much storage for intermediate values. This is not an issue when computations are performed locally, but for distributed computations this is just as costly as doubling the working precision. A well-known correction factor, attributed in [CGL83] to A. Björk, albeit also proposed by [Nee66], greatly improves the accuracy of the two-pass algorithm when computing the variance. In [PTKB16] we generalized this correction factor to moments of arbitrary order. Our scheme transmits only one additional value in the second pass, but can correct for the error in moments of all orders, providing increased accuracy for higher order moments at a fraction of the cost of generic compensation schemes. We provided numerical results for most of these new formulas, including comparisons with other formulations and an application to a scientific use case. In particular, we empirically observed that our generalized incremental and pairwise algorithms perform almost as well as the two-pass algorithm, and in some cases even better.

We now briefly recall the parallel update formulas that are relevant to the current study. For details about these, please refer to [PTKB16]. For  $p$  a non-negative integer and using  $E[\cdot]$  to denote the expectation, the  $p$ -th central moment of a (univariate, real) random variable  $X$  is defined as

$$\mu_p \triangleq E[(X - E[X])^p] , \quad (2.1)$$

when the expectations exist (some random variables, e.g. those with a Cauchy distribution, do not have an expectation). For a finite population of  $n$  equiprobable values in a multiset  $S = \{x_i\}_{i=1}^n$ ,

this reduces to

$$\mu_p = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^p \quad (2.2)$$

where  $\bar{x} \triangleq \frac{1}{n} \sum_{i=1}^n x_i$  is the mean. Note that  $S$  has to be a multiset, not a set, in order to allow for multiple copies of the same value.

The first central moment is exactly zero, and the second central moment is, by definition, the *variance*:  $\sigma^2 \triangleq \mu_2$ . We only consider the statistics of finite populations taken in their entirety, i.e., not sampled, to avoid issues of estimation bias. If  $S$  is instead just a finite sample of an infinite population, one may obtain unbiased estimates of the moments of the whole population [Hal46]. However, unbiased estimates of the moments do not, in general, lead to unbiased estimates of derived descriptive statistics such as standard deviation, skewness, and kurtosis.

Partitioning  $S$  into multisets  $\mathcal{A}$  and  $\mathcal{B}$  of respective sizes  $n_{\mathcal{A}}$  and  $n_{\mathcal{B}}$  and define  $\mu_{p,\mathcal{A}}$ ,  $\mu_{p,\mathcal{B}}$ ,  $\bar{x}_{\mathcal{A}}$ , and  $\bar{x}_{\mathcal{B}}$  to be the corresponding statistic computed over each partition, we defined

$$\begin{aligned} \delta_{\mathcal{B},\mathcal{A}} &\triangleq \bar{x}_{\mathcal{B}} - \bar{x}_{\mathcal{A}} , \\ M_p &\triangleq n\mu_p , \end{aligned}$$

and again give  $M_p^{\mathcal{A}}$  and  $M_p^{\mathcal{B}}$  an equivalent definition restricted to each partition. We will find it more convenient to work with these  $M_p$  quantities, or *aggregates*, rather than  $\mu_p$ , though either may be readily obtained from the other. With these notations, we derived the following pairwise arbitrary-order and arbitrary set decomposition update formulas:

$$\bar{x} = \bar{x}_{\mathcal{A}} + \frac{n_{\mathcal{B}}}{n} \delta_{\mathcal{B},\mathcal{A}} ,$$

and or any integer  $p \geq 2$ ,

$$\begin{aligned} M_p &= M_p^{\mathcal{A}} + M_p^{\mathcal{B}} + n_{\mathcal{A}} \left( \frac{-n_{\mathcal{B}}}{n} \delta_{\mathcal{B},\mathcal{A}} \right)^p + n_{\mathcal{B}} \left( \frac{n_{\mathcal{A}}}{n} \delta_{\mathcal{B},\mathcal{A}} \right)^p \\ &\quad + \sum_{k=1}^{p-2} \binom{p}{k} \delta_{\mathcal{B},\mathcal{A}}^k \left[ M_{p-k}^{\mathcal{A}} \left( \frac{-n_{\mathcal{B}}}{n} \right)^k + M_{p-k}^{\mathcal{B}} \left( \frac{n_{\mathcal{A}}}{n} \right)^k \right] . \end{aligned}$$

A number of commonly used statistics can be derived from the  $M_p$  aggregates such as, for instance, the *sample variance*

$$\sigma_x^2 = \frac{M_2}{n}$$

and the *unbiased sample variance*, cf. [DCD86]

$$s_x^2 = \frac{M_2}{n-1} .$$

Higher-order aggregates allow for the computation of various *skewness* and *kurtosis* estimators, such as, respectively:

$$g_1 = \frac{M_3}{n s_x^3}$$

and

$$g_2 = \frac{M_4}{ns_x^4}.$$

In our experiments we typically report *kurtosis excess*,  $g_2 - 3$ , rather than raw kurtosis, for the kurtosis of a normal distribution is 3 which allows one to distinguish between *platykurtic* (thin tails; kurtosis defect) and *leptokurtic* (fat tails; kurtosis excess) distributions.

## 2.5 Evaluation Criteria

In order to assess parallel scalability with our various programming models and implementations, we used the following evaluation criteria:

1. *Strong scaling*, i.e., at constant total work, also known as relative speed-up. Denoting  $T_N(p)$  the wall clock time measured to execute the calculation, strong scaling is defined as:

$$S_N(p) = \frac{T_N(1)}{T_N(p)}.$$

Some authors prefer to write the numerator as  $T_s$  rather than  $T_N(1)$  to make it clear that the parallel algorithm should be compared to the most efficient serial implementation available and not just the parallel algorithm run on a single task.

Evidently, optimal (linear) scaling is attained when  $S_N(p) = p$  and, therefore, strong scaling results can be visually inspected by plotting  $S_N$  versus the number of tasks: optimal scaling is revealed by a line, the angle bisector of the first quadrant.

2. *Weak scaling*, i.e., at constant work per task, also known as rate of computation scalability. The rate of computation is defined as:

$$r(p) = \frac{N(p)}{T_{N(p)}(p)},$$

where  $N(p)$  now varies with  $p$ . Weak scaling is then measured by normalizing the rate of computation by that which is obtained with a single task. In particular, if the sample size is made to vary in proportion to the number of tasks, i.e., if  $N(p) = pN(1)$ , then

$$R(p) = \frac{r(p)}{r(1)} = \frac{pT_{N(1)}(1)}{T_{pN(1)}(p)} = \frac{pT_{N(1)}(1)}{pT_{N(1)}(p)} = \frac{T_{N(1)}(1)}{T_{N(1)}(p)}.$$

Therefore, optimal (linear) scaling is attained with  $p$  tasks when  $R(p) = p$ . Note that without linear dependency between  $N$  and  $p$ , the latter equality no longer implies optimal scalability.

Parallel scalability can thus also be visually inspected by plotting the values of  $R$  versus the number of tasks, and in this case also, optimality corresponds to the angle bisector of the first quadrant.

The information about time spent by each of the considered tasks can be extracted from the profiling output of the Legion implementations, by means of the Python script called `legion_prof.py`, which comes with the distribution of Legion. This script parses the performance output that is generated when the appropriate option flags are passed to the executable through the command line.

### 3 Parallel Statistics in Legion: Aggregation Regions Approach

We now explain the methodology which we used in order to re-formulate in an AMT context our bulk-synchronous SPMD implementation of the Learn/Derive/Pattern. Retaining this flow of operations is important because the original goal of meeting the predominant types of data analysis work flows remain. Furthermore, as will be seen below, the decomposition of the analysis workflow into independent operations, isolating those parts that do not require global communication from the others, allows for a natural and elegant transition to an AMT model.

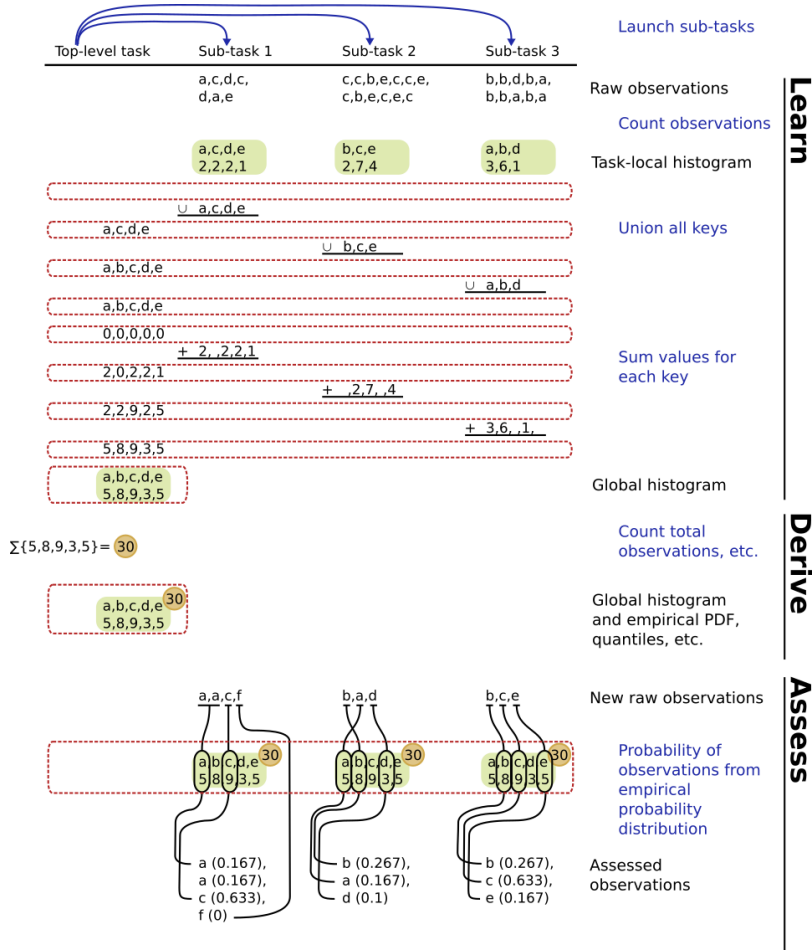
#### 3.1 Overview of the Aggregation Regions Approach

As described in §2, our SPMD models can take two different incarnations, depending on whether the analysis pertains to *moment-based* or *quanta-based* statistics. In the former case, the implementation of the Learn phase relies on an AllGather MPI collective to reduce the distributed local models into the global primary model whereas, in the latter case, this reduction uses a Gather-Broadcast two-step process, for the sake of reducing inter-process communication as the size of local models is not guaranteed to be negligible.

In the AMT model however, it is no longer necessary to express data movement explicitly as is done in the SPMD model, for instance with the dashed red arrow in Figure 3. Considering the same example for the sake of illustration, we can see that task parallelism can readily replace the tasks performed by the distributed processes. Instead of specifying movement of data from processes to a number of reduction nodes, it is instead sufficient to define a logical region of data where the equivalent of MPI collectives will be performed. The separation of the logical from the physical representation of data inherent to Legion makes it especially apt at supporting this purely data-driven scheme, which should work, albeit maybe not efficiently, with any generic mapper.

Our proposed AMT algorithm for the Learn/Derive/Assess work flow therefore replaces communication with a logical region that contains all model information, both primary and derived, which we call the *aggregation region*. Sub-tasks launched by a top-level task pick up work on those data segments to which they are assigned, in a similar manner to what is done by parallel processes in the SPMD context, at least for the Learn and Assess phases. However, a first noticeable difference is that no communication of data is expressed (even though it will occur under the hood; how this happens is entirely the responsibility of the run-time system). Instead, each task aggregates the primary statistics it has computed over its data segment, with those already stored in the aggregation region, changing them in-place. As a result, no broadcast of the global primary model is necessary, for all tasks using it (for instance, a new set of Assess sub-tasks) will directly access the aggregation region to retrieve the values of the statistic. Moreover, the Derive operation is now performed by the top-level task, for its results, also stored in the aggregation region, will be also logically available to any Assess tasks launched from the top level. This asynchronous many-task Learn/Derive/Assess scheme is represented in Figure 4, again for the case of order statistics.

Note that another benefit of this AMT model is that it allows us to unify the two SPMD imple-



**Figure 4.** A simplified example illustrating the operations of the task-based order statistics; solid blue arrows indicate task launches, dashed red rectangles represent the logical aggregation region. Sub-tasks of the top-level task are not obligated to complete in this order, as both union and addition operators are commutative.

mentations (AllGather and Gather-Broadcast) into a single paradigm, valid for both quanta-based and moment-based statistics. Also, because all aggregation operations (set unions, number additions and multiplications) are commutative, the learned primary model is guaranteed to be independent of the order in which tasks report their results. However strict locks must be enforced to prevent incomplete model updates: once a task  $t_i$  has read the values in the aggregation region, no other task can access this region before  $t_i$  has written its own results there.

For example, in the case of *in situ* descriptive statistics for MiniAero/Legion, the sub-tasks (numbered 1, 2, and 3 in Figure 4) are implemented as a method templated on the type of the Legion *accessor* used by the run-time to access data within a physical region. Legion pointers do not directly reference data, but instead name an entry in an index space, and are used when

accessing data within accessors for logical regions. In this setting, the accessor is specifically associated with the field being accessed and the pointer names the row entry. Since pointers are associated with index spaces they can be used with an accessor for a physical instance. In this way Legion pointers are not tied to memory address spaces or physical instances, but instead can be used to access data for any physical instance of a logical region created with an index space to which the pointer belongs.

Given a `LegionAccessorType`, a concrete instance of the *in situ* statistics class provides an implementation of the the Learn operation of the descriptive statistics engine. In this case, the primary statistical model contains the following 8 quantities, stored in double precision: sample size, minimum, maximum, mean, and centered  $M_2$ ,  $M_3$  and  $M_4$  aggregates as defined in §2.4. The computation of the process-local model is done using the *online* versions of the respective update formulas for the quantities above, whereas the aggregation with the global model is computed by means of the *pairwise* versions, cf. [PTKB16] for details.

In turn, this Learn function is responsible for setting the `Legion IndexLauncher` that spawns as many sub-tasks as requested. These operate on the partitioned input data and are executed by the run-time, as illustrated by the blue arrows on top of Figure 4. The Learn function also creates the *Legion region requirements* needed by the sub-task launcher; these are Legion objects used to describe the logical regions requested by launcher objects. They also convey the privileges and coherence that is requested on the specified logical region. In the case of a descriptive statistics Learn task, two region requirements are needed: one for the input data to be read (and only read), and the other for the output region where results are to be aggregated (and which therefore is requires both read and write access).

In addition, the *in situ* statistics class provides an implementation of the Derive operation, which is to be called only by the top-level task, for it needs only a single pass over the small set of primary statistics in order to compute the derived statistics, as illustrated in Figure 4. Namely, for descriptive statistics, these are the variance, standard deviation, skewness and kurtosis estimators. This operation is de facto negligible for it involves a dozen arithmetic operations computer on a single region, performed by a single task. Note that in this case the Legion region requirements are a read-only access to the first field of the logical region used to store the statistical model, and write permission for the second field of the same region.

We did not implement the Assess operation for the MiniAero/Legion experiment, for this study was focused on validating the proposed AMT design: because the Assess phase is purely task-local, it does not bring any additional information in this regard. It suffices to say here that the Assess method of the *in situ* statistics class is to be launched in as many sub-tasks as necessary, similarly to what is done for the Derive operation. For instance, in the case of descriptive statistics, every Assess task will mark each datum of the data subset to which it is assigned with its relative deviation with respect to the model mean and standard deviation (which amounts to the one-dimensional Mahalanobis distance from the value predicted by a Gaussian model [Mah36]).

### 3.2 *In Situ* Setting with MiniAero

We now discuss how the Legion version of MiniAero was interfaced with the in situ AMT statistics code described above. This required the use of adaptor code which we kept as small as possible; as this was a proof-of-concept implementation, it was not designed with consideration for genericity.

The first part of MiniAero/Legion to be modified is its `TaskIDs.h` file, whose `enum TaskIds` defines one index for each task allowed to be created at run-time. Specifically, those added for the sake of the in situ statistics are the following :

```
initialize_statistics_TID,  
learn_statistics_TID,  
derive_statistics_TID,  
dump_statistics_TID,
```

which respectively identify the statistics initialization, learn, derive and print-out tasks. In addition, the file `main.cc` must include the `Statistics.h` header, so that its main routine may register the tasks corresponding to the IDs listed above as follows:

```
MiniAero::Statistics::register_tasks();
```

We also modified the user interface of MiniAero/Legion in order to allow for the specification of additional parameters in the command line arguments of the MiniAero executable. Specifically, we added the 3 following instance variables to `Interface.h`:

```
bool descriptive_statistics;  
int statistics_frequency;  
int statistics_component;
```

to capture, respectively, whether the in situ statistics should be executed and, if so, how often and for which variable amongst the 5 components of the MiniAero solution vector. Note that, at this point, only one variable can be analyzed at a time, but this will be modified in subsequent versions. Furthermore, the implementation of the user interface in `Interface.cc` was modified, first for the constructor which now takes in 3 the additional parameters:

```
Interface::Interface()  
: problem_type(0), num_blocks(0),  
  blocks_x(0), blocks_y(0), blocks_z(0),  
  interval_x(0), interval_y(0), interval_z(0),  
  mesh_scale(1.0), output_frequency(1), time_steps(1),  
  length_x(0.0), length_y(0.0), length_z(0.0),  
  ramp_angle(0.0), dt(0.0), wait(false), viscous(false),  
  second_order_space(false), output_results(false),  
  descriptive_statistics(false), statistics_frequency(1), statistics_component(0)
```

with default values so that the analysis is *not* run unless the user explicitly so requests. The `enroll_options()` function is then modified by incorporating the 3 corresponding option settings:

```
options_.enroll("descriptive_statistics",GetLongOption::NoValue,
               "If specified, will compute in situ descriptive statistics.",
               NULL);

options_.enroll("statistics_frequency",GetLongOption::MandatoryValue,
               "Number of time steps after which a new statistical
               analysis is performed.",
               NULL);

options_.enroll("statistics_component",GetLongOption::MandatoryValue,
               "Component of solution vector upon which a statistical
               analysis is performed.",
               NULL);
```

which are also added to the parsing method `parse_options()` with, on one hand at the top of the function:

```
descriptive_statistics = options_.retrieve("descriptive_statistics");
```

and, on the other hand at the bottom of the function:

```
{
    const char *temp = options_.retrieve("statistics_frequency");
    if (temp != NULL) {
        statistics_frequency = std::strtol(temp, NULL, 10);
    }
}

{
    const char *temp = options_.retrieve("statistics_component");
    if (temp != NULL) {
        statistics_component = std::strtol(temp, NULL, 10);
    }
}
```

following the format already in place for the existing interface options. Consequently, 3 new optional command line arguments become available; for instance, the following:

```
-descriptive_statistics -statistics_frequency 20 -statistics_component 0
```

when passed to the MiniAero executable, results in the in situ analysis tasks being invoked (the presence of `-descriptive_statistics` results in the corresponding Boolean to become `true`) every 20 time-steps, for the first component of the solution vector.

The last part of MiniAero/Legion that requires adaptor code is `toplevel.cc`, which also must include `Statistics.h` so it can instantiate a `Statistics` object. Prior to that, the interface options related to the in situ statistics must be retrieved, which is done as follows:

```
bool do_stats = interface.descriptive_statistics;
int stat_freq = interface.statistics_frequency;
int component = interface.statistics_component;
```

in a similar way as what is done for the other interface options. Subsequent versions of the AMT statistics framework will design an interface with more options, modeled after those currently offered by the VTK parallel statistics engines. The `Statistics` object is then instantiated as follows:

```
MiniAero::Statistics statistics(meshdata, ctx, runtime, mesh->num_blocks_,
                               task_wait_all_results, component);
```

which must be done somewhere *before* the driver time-loop. This is in fact similar to what is done for the other main task objects of the Legion version of MiniAero (`bc`, `solver`, `flux`, etc.), and for this reason we placed the `Statistics` instantiation immediately after all those already present in `toplevel.cc` for these other objects. The aggregation regions defined in §3.1 must also be created, which in this experimental implementation is done explicitly as follows for primary statistics:

```
Rect<1>
primary_stat_rect(Point<1>(0), Point<1>(MiniAero::Statistics::n_primary_stat-1));
IndexSpace primary_stat_is
    = runtime->create_index_space(ctx, Domain::from_rect<1>(primary_stat_rect));
FieldSpace primary_stat_fs = runtime->create_field_space(ctx);
{
    FieldAllocator allocator
        = runtime->create_field_allocator(ctx, primary_stat_fs);
    allocator.allocate_field(sizeof(double), MiniAero::Statistics::primary_stat_FID);
}
LogicalRegion primary_stat_lr
    = runtime->create_logical_region(ctx, primary_stat_is, primary_stat_fs);
runtime->attach_name(primary_stat_lr, "primary_stat_lr");
```

and as follows for derived statistics:

```
Rect<1>
derived_stat_rect(Point<1>(0), Point<1>(MiniAero::Statistics::n_derived_stat-1));
IndexSpace derived_stat_is
    = runtime->create_index_space(ctx, Domain::from_rect<1>(derived_stat_rect));
FieldSpace derived_stat_fs
    = runtime->create_field_space(ctx);
{
    FieldAllocator allocator
        = runtime->create_field_allocator(ctx, derived_stat_fs);
```

```

    allocator.allocate_field(sizeof(double),MiniAero::Statistics::derived_stat_FID);
}
LogicalRegion derived_stat_lr
    = runtime->create_logical_region(ctx,derived_stat_is,derived_stat_fs);
runtime->attach_name(derived_stat_lr, "derived_stat_lr");

```

which will then be passed explicitly to the statistics invocation methods, *inside* the time-loop:

```

if (do_stats && !(n_s % stat_freq)) {
    statistics.initialize_statistics(primary_stat_lr);
    statistics.learn_statistics(MeshData::solution_n,primary_stat_lr);
    statistics.derive_statistics(primary_stat_lr,derived_stat_lr);
    statistics.dump_statistics(primary_stat_lr,derived_stat_lr);
}

```

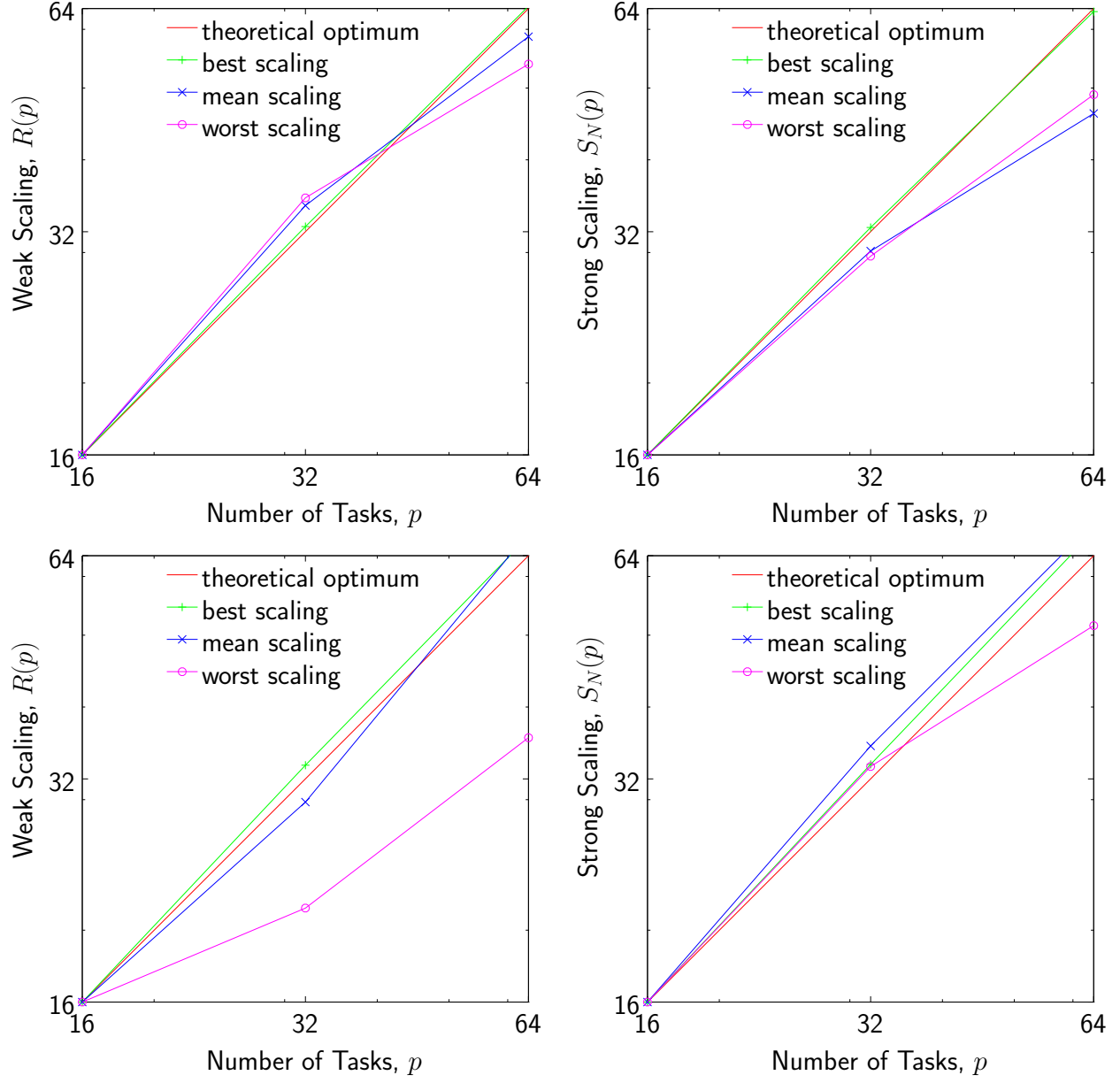
with the last call being optional, to be used only if a print-out of the computed statistics is sought. This concludes the description of the adaptor code that is needed at the time of writing, and which was used to test out the code.

### 3.3 Aggregation Regions Results

We also added timing and profiling capabilities to those already present in the code, in order to to measure performance of the *in situ* analysis framework. For instance, some results of weak and strong scalability series measured on 1 to 4 compute nodes of the experimental cluster shannon, fully subscribed with 16 Learn tasks per node, are presented here in Figure 5. Please note that more scaling experiments with the MiniAero/Legion setting are presented and discussed in detail in [PB16, PBH<sup>+</sup>16].

In the case presented here, each test was run 20 times, in order to capture any performance variability. As a result, three separate series of timings were retained, corresponding to the the shortest (“best”), average, (“mean”) and longest (“worst”) *in-situ* analysis execution speeds. In particular, these scaling plots now exhibited a surprisingly low worst-case weak scaling plot while, on the other hand, a rather robust strong scaling in all cases.

While the former finding can be easily explicated by the inherent nature of strong scaling, not having yet run its course of potential parallel gains as data size increased by a factor of 8, the latter is more difficult to attribute to a single cause (even though several could be suggested, especially in relation to the mean and worst cases super-scaling observed with smaller meshes). Further studies were thus warranted in order to definitely conclude in this regard, hereby confirming the need to explore other programming paradigms.



**Figure 5.** Scaling of Learn tasks on up to 4 nodes: left, weak scaling from a  $128^3$  (top) or  $256^3$  (bottom) grid; right, strong scaling with a  $128^3$  (top) or  $256^3$  (bottom) grid per task.

## 4 Parallel Statistics in Legion: the Legion-SPMD Approach

### 4.1 Overview of the Legion-SPMD Approach

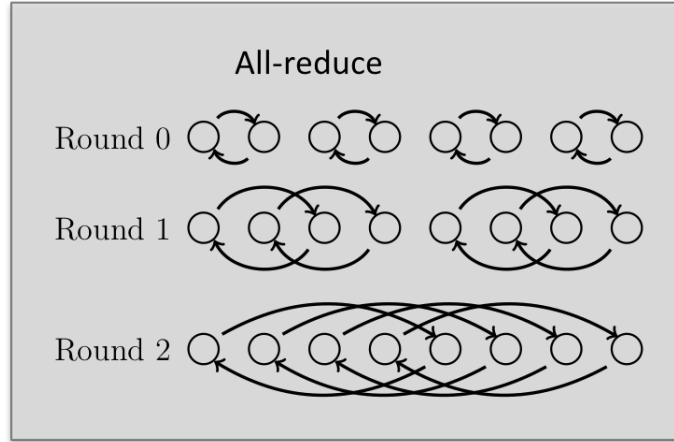
The aggregation region approach, described in the previous section, presents an elegant solution that abides by a SPMD implementation while still benefiting from any asynchronous execution. However there are a few drawbacks to this approach that could potentially be improved upon: it represents an *all-to-one* model of collectives where all contributions to a collective are funneled to one-point (the aggregation region), and the result is not immediately available in the tasks contributing to the collectives but rather to the parent task.

For instance, in Figure 4 the sub-tasks 1, 2, 3 launched during the Learn phase contribute to the global model, but will not have access to it during their respective lifetimes. The global model is logically available only in the top-level task at the end of the Learn phase. Moreover, any Assess phase tasks that use this global model will require a fresh SPMD launch. This fork-join model may be seen as a “pinching” of the SPMD approach preventing the creation of sub-tasks with perfect SPMD form, that can run through the entire lifetime of the application. Nearly all *in situ* analyses require some form of global communication and most applications themselves might require frequent collectives, which can make the aggregation region approach incur noticeable overhead.

Legion and similar systems provide a sequential abstraction. A program consists of a sequence of tasks, and the system is responsible for finding parallelism that respects the original program order. Because tasks obey a sequential ordering, they need to be analyzed in order. In Legion, this currently happens on a single node, leading to a performance bottleneck. In general, if it is intended to launch  $N$  tasks where  $N$  is a function of the number of nodes in the machine, the overhead of the run-time system will be  $O(N)$ . This bound is inherent given the chosen abstractions and even a distributed analysis of tasks suffers from this problem. For example, StarPU [AAF<sup>+</sup>16] follows an approach where each node independently filters the set of tasks to just those to be executed by that node. But because every node must still consider all tasks, the overall cost of this approach is  $O(N)$  as in Legion.

A proper solution—which achieves  $O(1)$  analysis cost per node—requires abstractions that are independent of the size of the machine. Legion already provides this via *index launches*, which describe a set of tasks to be executed in parallel on the machine, although the current run-time implementation does not fully take advantage of this. Regent, a compiler for the Legion programming model, can automatically *control replicate* these index launches to produce long-running tasks called *shards* that amortize the cost of this analysis. Critically, the compiler can determine from the sequence of index launches the required patterns of communication for the application on a distributed-memory machine, and can automatically generate the appropriate code for that communication with no additional user input. Regent control replication has been demonstrated to achieve good scalability to 1024 nodes for a variety of structured and unstructured applications. An implementation for the Legion run-time itself is also in progress [SLT<sup>+</sup>17].

We propose herein an alternative model for collectives that aligns exactly with the SPMD philosophy, while removing this all-to-one funneling effect. Moreover, the collective result will be available within each of the SPMD sub-tasks launching the collectives. This model is “MPI-like” and relies on the binary-tree algorithm that has a  $\log_2(N)$  scaling on the number of SPMD elements. In this model the control is passed down from the top-level task to the SPMD sub-tasks which have access to the global model once it has been properly Derived. The Assess phase can then happen inside these sub-tasks since the global model will be available locally to each of them. In other words the sub-tasks in Fig. 4 can persist through the Learn/Derive/Assess phases without requiring a break for the Derive phase followed by a fresh SPMD launch for the Assess phase.



**Figure 6.** Pictorial representation of the binary-tree algorithm for performing an AllReduce collective over 8 SPMD elements.

This model is designed to provide MPI-like functionality, where an AllGather or AllReduce is called from within each rank, but the result becomes available to every participating rank. Accordingly, each SPMD sub-task first performs the Learn operation on the portion of the data it owns. It then launches a *collectives task* from within itself which performs the following sequence of operations:

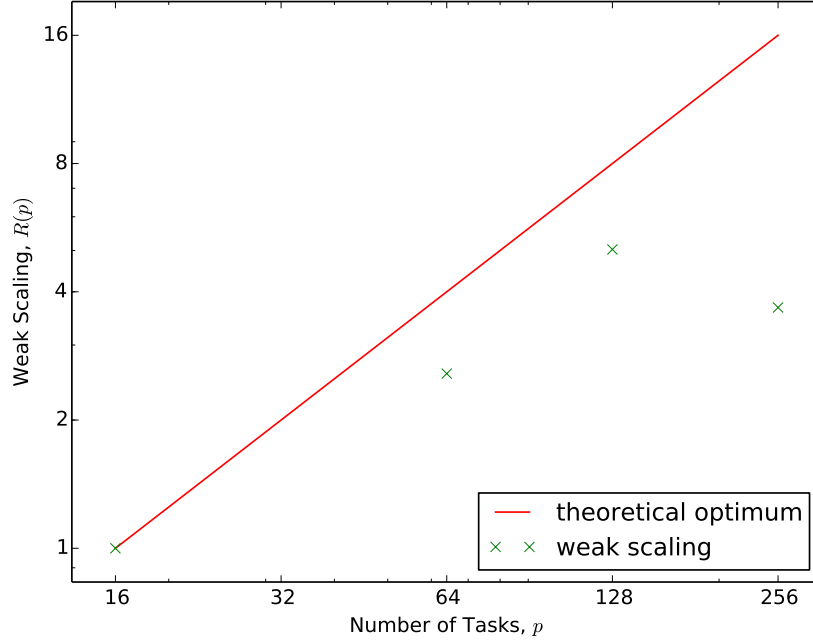
- Launch *collectives round-tasks*, the number of rounds being equal to  $\log_2(N)$  where  $N$  is the total number of SPMD elements.
- Execute the *collectives round-tasks* in sequence, each of which involves communication with exactly one other SPMD element.
- The index of the SPMD element to communicate with in each *collectives round-task* is also fully pre-determined from  $N$ .
- Each collective round-task involves updating the global model in place and the binary tree algorithm guarantees that this would be the true global model at the end of the last *collectives round-task*.

The collectives rounds are pictorially represented in Fig. 6 for the case of 8 SPMD elements performing an AllReduce.

To enable such a functionality we have implemented a light-weight collectives library that implements the binary-tree algorithm. The library allows any generic application with SPMD tasks to launch a *collectives task* from within each SPMD element. The launching of the collectives round-tasks, their execution in sequence, the communication with the predetermined partner in each round, and the update of the result in place are all orchestrated using Legion Phase Barriers. The Phase Barriers were designed to be light-weight synchronization primitives and can be created in large numbers without incurring much performance overhead. The actual reduction operation itself is a virtual method that provides flexibility over the type of data being reduced over, and the type of the reduction operation itself.

To assess the scalability of this approach we have performed weak scaling experiments of a stand-alone collectives prototype that emulates an AllReduce operation over a skeletal SPMD application on Titan, a Cray platform described in §4.3. The prototype code does not have tasks doing meaningful computations, but it simply performs an index launch of the collectives task over a certain number of SPMD shards. The collectives task within each shard then launch further sub-tasks to perform the rounds described above. Each shard is effectively contributing one floating point number to the AllReduce, and hence the result is also a floating point number. This is therefore a very conservative experiment that simply exposes the overhead for setting up the phase barriers and performing the sub-tasks in the designated order. In the setting of an actual application composed of tasks performing meaningful computations, the communication costs for the collectives rounds and the overhead for ordering the collectives rounds tasks in a sequence will be efficiently hidden by the Legion run-time. However, our prototype was designed specifically in order to *not* hide behind such effects so that scalability of the approach might be assessed in a stringent manner.

Figure 7 shows results of weak scaling experiments of the collectives prototype on Titan. Each run was performed with one legion process and one SPMD shard task per node on Titan. The results show that the scalability is reasonably good, close to the ideal scaling line, up to 128 nodes. However, when reaching 256 nodes a noticeable departure from ideal scaling occurs. Our experiments showed that as the number of nodes increased, the completion time for the collectives varied dramatically. In order to minimize possible effects due to unfavorable node placements by the batch system, the experiment was run successively 10 times in one single batch job (while the set of nodes remained constant), and even then the variation was significant (up to an order of magnitude). This suggests that the experiment is sensitive to network variability. Note that the fastest recorded time for the 256 nodes case was 0.47 seconds, indicating that these were extremely short experiments making them very susceptible to network variability. We are currently studying the scalability of the prototype in a variability-free environment on a dedicated cluster to assess the weak scalability better.



**Figure 7.** Weak scaling of AllReduce collective tasks.

## 4.2 In Situ Setting with S3D

In order to conduct scalability studies independently of load-balancing issues, we applied the *in situ* descriptive statistics engine to simulation results obtained using the massively parallel, compressible DNS code S3D [HSSC05].

The configuration investigated consists of a temporally evolving jet between *n*-dodecane and diluted air with  $X_{O_2} = 0.15$  and  $X_{N_2} = 0.85$ . The ambient gas temperature is  $T_{amb} = 960$  K, the fuel temperature is  $T_{fuel} = 450$  K, and the operating pressure is  $p = 25$  bar. The fuel stream is initialized with a mixture fraction  $\xi$  of 0.35: this value of  $\xi$  was chosen to be higher than those usually observed in practical diesel sprays downstream of the evaporation region in order to guarantee the existence of flow regions containing the most reactive composition at the time of ignition. The mean velocity of the fuel jet is  $u_j = 21$  m/s, the slot half-width is 0.25 mm, and the jet Reynolds number is  $Re_j = 6600$ . Reaction rates were computed using a 35 species reduced kinetics mechanism including both the low- and high-temperature oxidation pathways of *n*-dodecane.

The full simulation was conducted on a computational grid initially consisting of  $1200 \times 1500 \times 1000$  grid points in the stream-wise, transverse and span-wise directions respectively. The grid spacing is uniform in the stream-wise and span-wise directions and equal to  $\Delta x = 3 \mu m$ . To reduce the total grid count, a discretization with a uniform central region and stretched edges was used in the transverse direction: within the uniform region, the grid spacing is  $3 \mu m$ . As the jet expanded along the in-homogeneous direction during its temporal evolution, the uniform central region was periodically enlarged. At the end of the simulation, the number of grid nodes in the

transverse direction was 2400. The velocity field was initialized by superposing turbulent fluctuations on top of a hyperbolic tangent mean velocity profile. The fluctuations were generated using a synthetic homogeneous isotropic turbulence spectrum, and chemical reactions were deactivated during the initial laminar-turbulence transition of the velocity shear layer.

### 4.3 Test Platform

The test platform with S3D-Legion is Titan, a Cray massively parallel computational cluster, located Oak Ridge National Laboratory and meant to be used by premier science applications. It was built by Cray with funding mostly from the United States Department of Energy. Titan features a hybrid architecture comprising 18,688 AMD Opteron central processing units (CPUs) and an equal number of Nvidia Tesla K20X graphics processing units (GPUs). It has been the first such platform with demonstrated peak performance above 10PFLOPS, at 17.59PFLOPS [TOP13].

Titan is dedicated to science applications, with a selective process being in place in order to decide which projects can run on it. It is especially worth noticing that S3D is amongst six such applications that have been identified, as early as 2011, in order to prepare for exascale-capable codes.

Because of the hybrid CPU/GPU architecture of Titan, existing science applications that had been historically developed in a CPU-only context had to undergo substantial changes. As a result, the version of S3D that was ported to Titan exhibits greater performance than the CPU-only version.

### 4.4 Results

Each test was also run 20 times in an attempt to account for performance variability across runs. As in §3.3, three separate series of timings were retained, corresponding to the the shortest (“best”), average, (“mean”) and longest (“worst”) *in-situ* analysis execution speeds for each of the considered test cases.

On one hand, Table 1 presents the results of a weak scalability ( $R(p)$ ) experiment for the descriptive Learn task, scaling up to 128 tasks on as many distinct compute nodes of Titan, with a constant load of  $32^3$  grid points per task.

On the other hand, Tables 2 and 3 show the results of strong scalability ( $S_N(p)$ ) experiments, also with always a single task per compute node on Titan, with constant global meshes of respective size  $64 \times 128 \times 64$  and  $256 \times 128 \times 128$ .

These results are summarized in Figures 8 and 9 for the average scaling series. These reveal optimal scaling, both weak and strong, across the considered example space. We acknowledge however that we could not approach the Amdahl limit for strong scaling, as it turned out that a problem size allowing to approach it for descriptive statistics Learn tasks would be too large

**Table 1.** Weak scaling of Learn tasks on Titan: ensemble of 20 runs with a  $32^3$  grid per task and a single task per node.

$p$	$N(p)$	best	mean	worst
		ms   $R(p)$	ms   $R(p)$	ms   $R(p)$
2	65536	2.74   1.00	2.79   1.00	3.02   1.00
4	131072	2.69   2.04	2.81   1.98	3.73   1.62
8	262144	2.69   4.07	2.72   4.10	2.97   4.06
16	524288	2.71   8.11	2.83   7.89	3.02   7.98
32	1048576	2.71   16.2	2.76   16.1	2.95   16.4
64	2097152	2.70   32.5	2.75   32.5	3.01   32.0
128	4194304	2.70   65.1	2.73   65.4	2.85   67.7

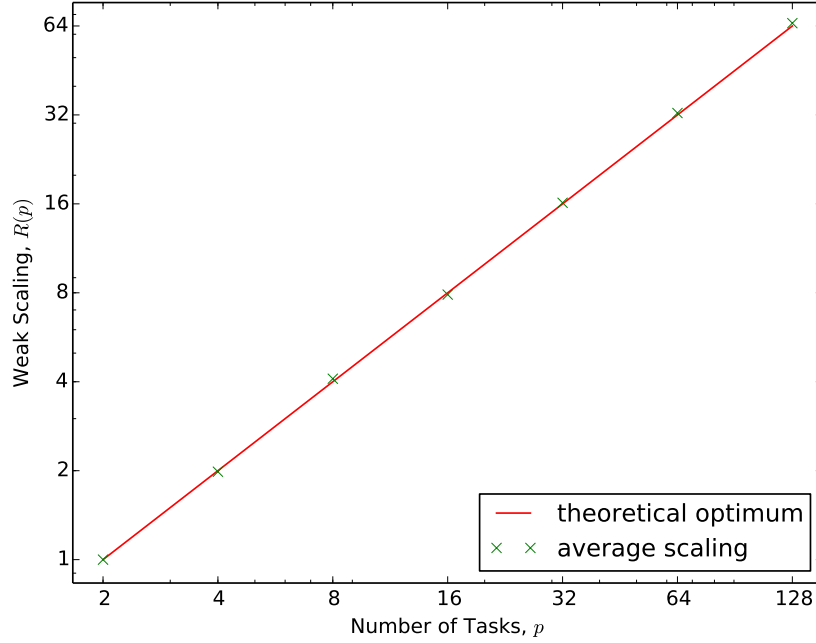
**Table 2.** Strong scaling of Learn task on Titan: ensemble of 20 runs with a  $64 \times 128 \times 64$  global grid and one task per node.

$p$	$N/p$	best	mean	worst
		ms   $S_N(p)$	ms   $S_N(p)$	ms   $S_N(p)$
2	262144	21.9   1.00	23.7   1.00	27.3   1.00
4	131072	11.2   1.95	11.7   2.02	12.4   2.20
8	65536	5.42   4.04	5.57   4.25	5.86   4.66
16	32768	2.72   8.05	2.78   8.53	2.98   9.16
32	16384	1.27   16.0	1.29   17.1	1.42   19.2

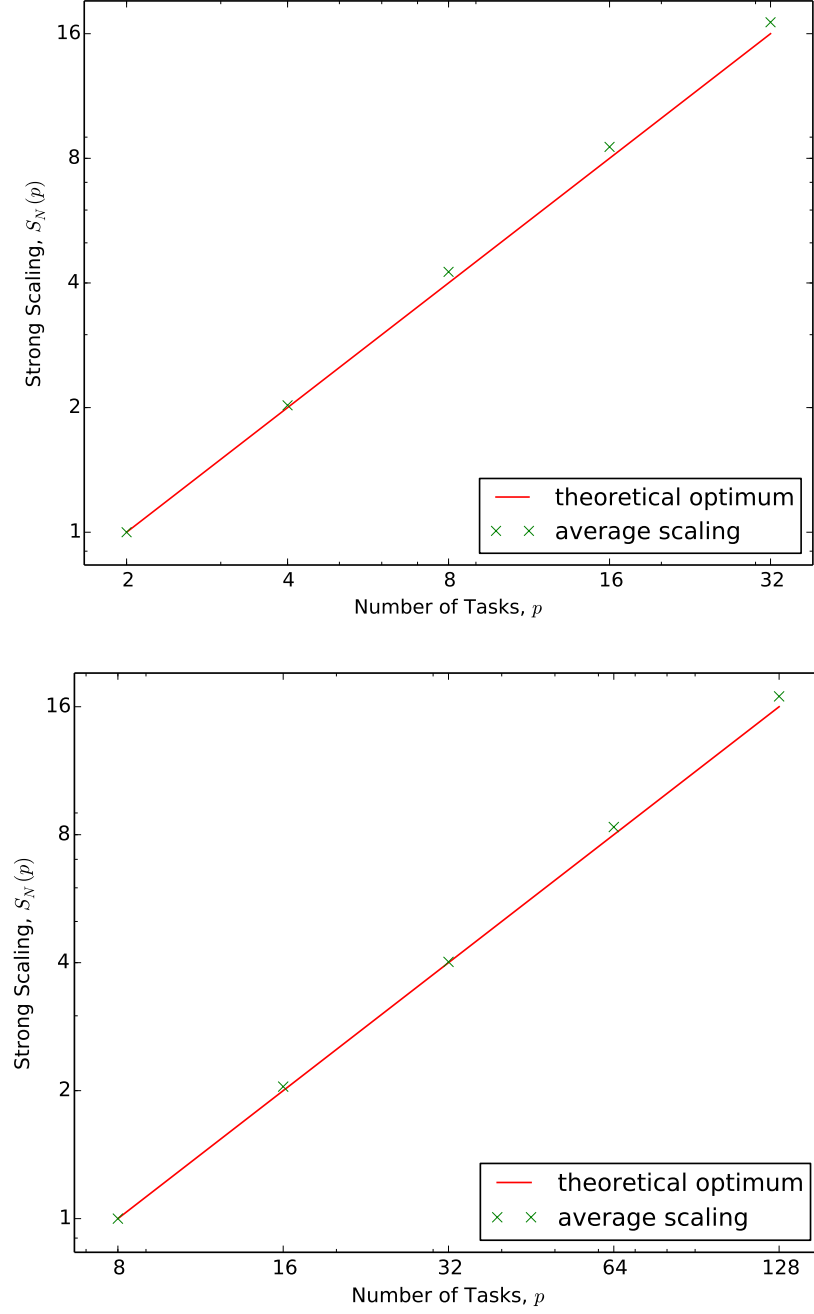
for the main processing computation. In other words, this finding demonstrates that our *in situ* implementation is more scalable than the overall application and therefore does not impact its scalability.

**Table 3.** Strong scaling of Learn tasks on Titan: ensemble of 20 runs with a  $256 \times 128 \times 128$  global grid and one task per node.

$p$	$N/p$	best	mean	worst
		ms   $S_N(p)$	ms   $S_N(p)$	ms   $S_N(p)$
8	524288	43.9   1.00	46.2   1.00	48.5   1.00
16	262144	21.9   2.00	22.6   2.04	24.1   2.01
32	131072	11.0   3.99	11.5   4.02	12.4   3.91
64	65536	5.42   8.10	5.54   8.34	5.71   8.49
128	32768	2.70   16.3	2.73   16.9	2.85   17.0



**Figure 8.** Weak scaling of descriptive Learn tasks on Titan, averaged over an ensemble of 20 runs with a  $32^3$  grid per task, and a single task per node.



**Figure 9.** Strong scaling of descriptive Learn tasks on Titan, averaged over an ensemble of 20 runs, with a  $64 \times 128 \times 64$  (top) and  $256 \times 128 \times 128$  (bottom) grid and a single task per node.

## 5 Conclusion

In conclusion, the results herein are very promising, and the statistics package is planned for deployment *in situ* at-scale in upcoming S3D science runs. Moreover, the integration work with Legion-S3D and the development of the lightweight collectives library can be modeled and directly leveraged by UDDAP team members at University of Utah, LANL, and Kitware.

Planned future work includes integrating the *in situ* kernels developed at these other institutions into a more comprehensive and complex workflow. In particular, we are currently working on integrating *order statistics* as another analysis kernel for S3D-Legion, in order to provide scientists with the ability to derive empirical probability density functions to, in particular, evince outlying results and deviations from models computed by the already integrated descriptive statistics engine.

## References

- [AAF<sup>+</sup>16] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. Technical Report RR-8927, INRIA Bordeaux Sud-Ouest, 2016.
- [BGP<sup>+</sup>09] J. Bennett, R. Grout, P. P. Pébaÿ, D. Roe, and D. Thompson. Numerically stable, single-pass, parallel statistics algorithms. In *Proc. 2009 IEEE International Conference on Cluster Computing*, New Orleans, LA, U.S.A., August 2009.
- [BTSA12] M Bauer, S Treichler, E Slaughter, and A Aiken. Legion: expressing locality and independence with logical regions. In *SC '12: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.
- [CGL83] Tony F. Chan, Gene H. Golub, and Randall J. LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, 37(3):242–247, August 1983.
- [DCD86] D. Dacunha-Castelle and M. Duflo. *Probability and Statistics*, volume 1. Springer-Verlag, 1986.
- [DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, December 2004.
- [Edd10] Dirk Eddelbuettel. High-performance and parallel computing with R, 2010.
- [FFL15] K. J. Franko, T. C. Fisher, and P. Lin. CFD for next generation hardware: Experiences with proxy applications. In *Proc. 22<sup>rd</sup> AIAA Computational Fluid Dynamics Conference*, Dallas, TX, U.S.A., June 2015.
- [Gro09] W. Gropp. MPI at exascale: Challenges for data structures and algorithms. In *Proc. 16<sup>th</sup> European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, page 3. Springer-Verlag, 2009.
- [Hal46] Paul R. Halmos. The theory of unbiased estimation. *The Annals of Mathematical Statistics*, 17(1):34–43, March 1946.
- [HSSC05] Evatt R Hawkes, Ramanan Sankaran, James C Sutherland, and Jacqueline H Chen. Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models. *Journal of Physics: Conference Series*, 16(1):65, 2005.
- [LL07] Philippe Langlois and Nicolas Louvet. How to ensure a faithful polynomial evaluation with the compensated horner algorithm. In *Proc. 18<sup>th</sup> Symposium on Computer Arithmetic (ARITH'07)*, pages 141–149, Montpellier, France, June 2007.
- [Mah36] P. C. Mahalanobis. On the generalised distance in statistics. *Proc. of the National Institute of Science of India*, (12):49–55, 1936.

- [Nee66] Peter M. Neely. Comparison of several algorithms for computation of means, standard deviations and correlation coefficients. *Communications of the ACM*, 9(7):496–499, July 1966.
- [ORO05] Takeshi Ogita, Siegfried M. Rump, and Shin’ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, November 2005.
- [PB15] P. P. Pébaÿ and J. Bennett. An asynchronous many-task implementation of *in situ* statistical analysis using Legion. Sandia Report SAND2015-10345, Sandia National Laboratories, November 2015.
- [PB16] P. P. Pébaÿ and J. Bennett. Scalability of asynchronous many-task *In Situ* statistical analysis on experimental clusters: Interim report. Sandia Report SAND2016-1487, Sandia National Laboratories, February 2016.
- [PBH<sup>+</sup>16] P. P. Pébaÿ, J. Bennett, D. Hollmann, S. Treichler, P. McCormick, C. Sweeney, H. Kolla, and A. Aiken. Towards asynchronous many-task *in situ* data analysis using Legion. In *Proc. 30<sup>th</sup> IEEE International Parallel & Distributed Processing Symposium, High Performance Data Analysis and Visualization Workshop*, Chicago, IL, U.S.A., May 2016.
- [PTBM11] P. P. Pébaÿ, D. Thompson, J. Bennett, and A. Mascarenhas. Design and performance of a scalable, parallel statistics toolkit. In *Proc. 25<sup>th</sup> IEEE International Parallel & Distributed Processing Symposium, 12<sup>th</sup> International Workshop on Parallel and Distributed Scientific and Engineering Computing*, Anchorage, AK, U.S.A., May 2011.
- [PTKB16] P. P. Pébaÿ, T. B. Terriberry, H. Kolla, and J. Bennett. Numerically stable, scalable formulas for parallel and online computation of higher-order multivariate central moments with arbitrary weights. *Computational Statistics*, 31(4):1305–1325, 2016.
- [SLT<sup>+</sup>17] E. Slaughter, W. Lee, S. Treichler, W. Zhang, M. Bauer, G. Shipman, P. McCormick, and A. Aiken. Control replication: Compiling implicit parallelism to efficient SPMD with logical regions. 2017. In Submission.
- [SME<sup>+</sup>09] M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann. State-of-the-art in parallel computing with R. Technical Report 47, Department of Statistics, University of Munich, 2009.
- [Sta10] Stata. Stata/MP performance report. Technical report, StataCorp LP, June 2010. version 2.0.0.
- [TOP13] TOP500. Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x, 2013.
- [WBS08] B. Wylie, J. Baumes, and T. Shead. Titan informatics toolkit. In *IEEE Visualization Tutorial*, Columbus, OH, October 2008.

- [WTP<sup>+</sup>08] M. H. Wong, D. C. Thompson, P. P. Pébař, J. R. Mayo, A. C. Gentile, B. J. Debusschere, and J. M. Brandt. OVIS-2: A robust distributed architecture for scalable RAS. In *Proc. 22<sup>nd</sup> IEEE International Parallel & Distributed Processing Symposium*, Miami, FL, April 2008.

## DISTRIBUTION:

2	MS 9018	Central Technical Files, 8944
1	MS 0899	Technical Library, 9536



